Cautious Adaptation of Defiant Components

Conference or Workshop Item

Cautious Adaptation of Defiant Components

Paulo Henrique Maia*, Lucas Vieira*, Matheus Chagas*, Yijun Yu†, Andrea Zisman†, and Bashar Nuseibeh†
*State University of Ceará, Fortaleza, CE, Brazil
pauloh.maia@uece.br, {lucas.vieira, matheus.chagas}@aluno.uece.br
†The Open University, Milton Keynes, United Kingdom
{yijun.yu, andrea.zisman, bashar.nuseibeh}@open.ac.uk

***Abstract*—Systems-of-systems are formed by the composition of independently created software components. These components are designed to satisfy their individual requirements, rather than the global requirements of the systems-of-systems. We refer to components that cannot be adapted to meet both individual and global requirements as "defiant" components. In this paper, we propose a "cautious" adaptation approach which supports changing the behaviour of such defiant components under exceptional conditions to satisfy global requirements, while continuing to guarantee the satisfaction of the components' individual requirements. The approach represents both normal and exceptional conditions as scenarios; models the behaviour of exceptional conditions as wrappers implemented using an aspect-oriented technique; and deals with both single and multiple instances of defiant components with different precedence order at runtime. We evaluated an implementation of the approach using drones and boats for an organ delivery application conceived by our industrial partners, in which we assess how the proposed approach helps achieve the system-of-systems' global requirements while accommodating increased complexity of hybrid aspects such as multiplicity, precedence ordering, openness, and heterogeneity.**

***Index Terms*—Defiant Component, Adaptation, Scenarios, Aspects**

I. Introduction

It is now 10:00am in busy London. The kidney transplant department in St Thomas' Hospital (Hospital A) has been informed that a matching kidney of a patient in Guy's Hospital (Hospital B) is now available for its patient waiting for surgery. The organ transplant department of both hospitals contact the SOSDronePayload company and make all the necessary arrangements for the organ to be transferred by drone from Hospital B to Hospital A. Drone_DR1235 is selected by SOSDronePayload to deliver the organ through its flying corridor, over the River Thames, to avoid land traffic. At a certain point during the journey the battery of Drone_DR1235 reduces dramatically and when it reaches 10%, Drone_DR1235 attempts to land before reaching its final destination (Hospital A). However, at this point, Drone_DR1235 was only 2 km away from Hospital A and, given the favourable wind conditions at the time, Drone_DR1235 could have reached Hospital A with its 10% battery capacity and delivered the critical organ.

In the scenario, *Drone_DR1235* was developed independently of the payload organ delivery application, and was not intended to change its behaviour during its execution, regardless of the application in which it is used. The above scenario is not unrealistic (in fact a custom-made drone delivered a kidney in April 2019 to a woman in Maryland [1]) and is likely to become increasingly common.

Many applications are developed by the composition of independently created software components into a single system. These systems are known as system-of-systems [1] (SoSs), and they support the execution of certain functionalities that cannot be achieved by individual participating components on their own. In such situations, it is necessary to support emergent behaviours that appear due to the combination of existing components into new larger systems, or even due to new requirements or contextual changes [2]. The participating software components have been designed to satisfy predefined requirements, which are called *local requirements*, following predefined specifications. They are not necessarily intended to change their behaviour during execution, nor to support some *global requirements*, i.e. complex functionalities that cannot be provided by individual participating components on their own, and that arise with the constitution of the system-of-systems. We refer to components that resist such changes, despite the need, as *defiant* components.

Dealing with defiant components in a system-of-systems is challenging since (i) one may not have access to the components' source code in order to change their behaviour, and (ii) the adaptation should occur only in *exceptional situations*, i.e, situations that are temporary and may be influenced by transitory environmental conditions, in order to meet the global requirements of the SoS. Existing requirement conflict resolution and adaptation approaches have not considered such defiant components [3]–[9].

In this paper we suggest a novel approach that considers the existence of different types of defiant components. We present a *cautious adaptation* approach which guarantees that changes in the behaviour of the components will not interfere with their original functionality, i.e., the local requirements of the participating components during their normal use, and will only be triggered in exceptional situations. For example, in the above scenario, the drone is considered a *defiant* component since it has been developed with a specification in which it should make a safe landing when its battery decreases to 10%. However, from the perspective of the payload organ delivery application, it should be possible to force the drone to continue flying, given the exceptional and favourable wind conditions, which will allow the drone to complete its journey.

[1] https://www.nytimes.com/2019/04/30/health/drone-delivers-kidney.html

The approach relies on the use of scenarios to model and identify exceptional situations, since the high abstraction level provided by scenarios facilitates identification and understanding of requirements, involvement of different stakeholders, and analysis of future events of an application. The use of scenarios has been demonstrated successfully in the literature to support behavioural model checking and to help with decision-making [10] [11] [12]. Furthermore, we assume that the source code of the components are not available and, therefore, we propose the use of *wrappers*, implemented using an aspect-oriented programming (AOP) technique [13], to support exceptional conditions formalised with behavioural semantics using LTS [14]. In our approach, exceptional situations are represented by *joinpoints* of scenario-based aspect weaving [15], given that they can handle exceptions.

Aspect-oriented programming (AOP) provides wrappers to allow for the introduction of changes into defiant components without requiring consent from the designer of the components, or access to the source code of the components, as in the case of our approach. AOP avoids the need to redesign a component to satisfy emergent behaviours, as in the case when using dependency injection techniques [16] or plugin-based approaches [17]. On the other hand, the use of AOP techniques can be risky, since changes introduced by the use of aspects may violate the original (local) requirements of the components. With the help of model checking, our work uses AOP in a way that guarantees the original requirements of the components under normal execution conditions.

The remainder of the paper is structured as follows. Section II describes an overview and a formalisation of the approach to support cautious adaptation of defiant components. Section III presents the detailed process of the cautious adaptation approach. Section IV describes the implementation and the evaluation of the work in a payload organ delivery application. Section V discusses related work. Section VI concludes the paper and discusses future work.

II. Approach Overview

Figure 1 presents an overview of the cautious adaptation approach. As shown in the figure, the process is divided into three phases, namely: (i) *defiant component identification*, (ii) *wrapper design and implementation*, and (iii) *runtime cautious adaptation*. The first two phases occur at design time, while the last phase occurs at run time.

The *identification of defiant components* (phase 1) is supported by the use of scenarios to model the system and to formally verify defiant behaviour of components. Scenarios can be used to represent both exceptional and normal situations of a system, and have been widely used for modelling *what-if* situations [18]. In general, scenarios assume the use of off-the-shelf software components, with predefined requirements and specifications. Our approach uses Message Sequence Chart [19] (MSC) specifications, a standard International Telecommunication Union (ITU) notation for describing the interaction between communicating processes, and to model both ordinary and exceptional scenarios.

The *wrapper design and implementation* (phase 2) addresses the defiant behaviour of one or more components identified in phase 1, and uses aspect-oriented programming (AOP) [13] to implement wrappers. The use of wrappers is to support changes in the behaviour of the defiant components while keeping the satisfaction of the global system requirements. Exceptional situations identified from the scenarios in phase 1, indicate where changes in the system should occur. We call these places *joinpoints* and the identified joinpoints are represented as regular expressions as *pointcuts*. In this case, a replacement of the behaviour of a defiant component after joinpoints can be done through an aspect in terms of *advices*[2]. Advices handle exceptions by either invoking existing functionalities of the defiant components or introducing new functionalities and conditions to be executed. To be used correctly, the approach must enforce certain precedence order among the wrappers when more than one wrapper exists for the same exceptional condition.

The *runtime cautious adaptation* (phase 3) *weaves* the system with wrappers and executes the respective functionalities. This phase is based on the MAPE-K loop approach [20]. In this phase, monitors are used to observe contextual variables that characterise the surroundings of the system based on data collected from the environment (e.g., wind condition) or from sensors (e.g., Global Positioning Systems, accelerometer, battery level). The approach analyses the exceptional situations defined in phase 2. When exceptional situations exist, the wrapper is invoked and the exceptional functionalities, as implemented by the aspect advices, are executed.

The general problem being tackled can be formalised using the semantics of Problem Frames [21], as follows.

At the design time of a component (c), given its world context (W_c), its specification (S_c) must satisfy its requirements (R_c), i.e., $W_c, S_c \models R_c$. Suppose that component c is being used as part of a system-of-systems (s). Based on the analysis of exceptional scenarios concerning a specific context of the system (W_s), which satisfies the condition $W_s \implies W_c$, component c cannot satisfy global requirements of s. In other words, $W_s, S_s \not\models R_s$, where S_s contains S_c.

To verify that the wrapper executes its purpose, we formalise three conditions to be checked as follows:

$$W_s, S_s \not\models R_s \wedge W_s \implies W_c \quad \text{(defiant component identification)}$$
$$W_s, S_s|_{c \to w(c)} \models R_s \quad \text{(defiant behaviour removal)}$$
$$W_c \setminus W_s, S_{w(c)} \models R_c \quad \text{(safety assurance)}$$

- Defiant component identification checks whether the component's behaviour satisfies the global requirements.
- Defiant behaviour removal checks that after wrapping up the component with a new behaviour, the global requirement can be restored in exceptional situations.
- Safety assurance checks that after wrapping up the defiant component with a new behaviour, the local requirements are still satisfied in normal situations.

[2]In AOP, *joinpoints* are points of control where it is possible to add additional behaviour, while *pointcuts* refer to predicates that match joinpoints and *advices* are methods associated with pointcuts.

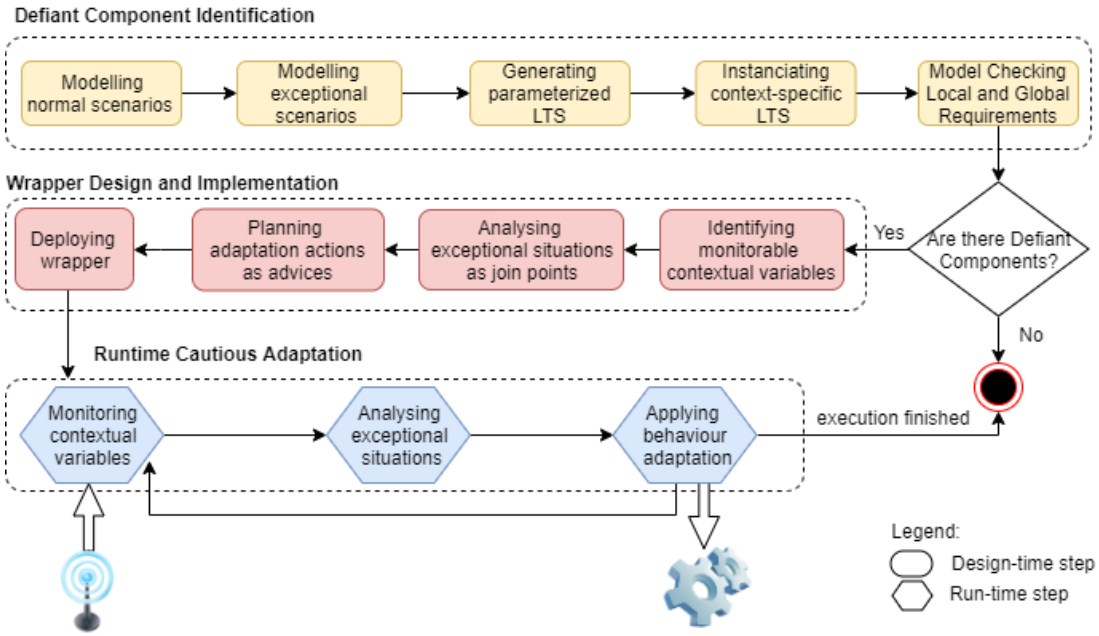

Fig. 1. Overview of the cautious adaptation approach in a workflow diagram

The introduction of a *wrapper* $w(c)$ changes the defiant behaviour of the component c, i.e. $W_s, S_{w(c)} \not\models R_c$, retaining the essential satisfaction of global requirements when c is substituted with $w(c)$: $W_s, S'_s \models R_s$ where $S'_s = S_s|_{c \to w(c)}$. Furthermore, the adaptation is *cautious* because other than the exceptional situation W_s, the wrapped component should behave like the originally designed component, i.e., $(W_c \setminus W_s), S_{w(c)} \models R_c$.

III. The Detailed Process of Cautious Adaptation

In this section we describe each phase of the approach (Figure 1) in details. We use the payload organ delivery example presented in Section I to illustrate the approach. Suppose two global requirements for the system as:

- GR1: the drone must take the payload organ from the sender hospital to the receiver hospital.
- GR2: in the case when the payload cannot be delivered, the drone should not lose it (e.g., by landing on the water).

A. Defiant Component Identification

As shown in Figure 1, the defiant component identification phase is composed of five steps. The first two steps consist of *modelling normal and exceptional scenarios* of a system-of-systems. The approach advocates the use of scenarios to model core functional requirements of the system-of-systems and local requirements for each component participating in the system. We assume that software engineers participating in these modelling activities are able of identifying exceptional scenarios regardless of the requirement elicitation techniques that are used during the process (see Section V).

Scenarios are described using Message Sequence Chart specification through two main syntactic components: *basic MSC* (bMSC) and *high-level MSC* (hMSC). A bMSC represents a set of asynchronous processes (or instances) with their exchanged messages. An hMSC consists of a graph of connected bMSCs representing parallel, sequential, iterating, and non-deterministic executions. In an hMSC graph the nodes refer to bMSCs and the edges represent execution sequences.

Figure 2 shows the hMSC specification of the payload organ delivery scenario. Each box in the hMSC corresponds to one bMSC, while the arrows represent the execution flow. The guard conditions in the bMSC transitions have to be satisfied in order to execute the functionalities represented in the next bMSC. Figure 3 shows parts of the scenarios of the bMSCs in Figure 2.

In Figure 2, the predefined scenarios are represented by full rectangles and are part of the normal specification of a drone. We assume that, with a controller, the pilot can control the drone to take off (bMSC *Take Off*). During a flight, a drone periodically checks the status (bMSC *Check Status*) of its internal devices and sensors such as battery level and distance from the destination, represented by b and d, respectively, in the guard conditions over the transitions. In the example, if the battery level is above the expected threshold $\theta = 10\%$, and the drone is not yet at its destination, the pilot can keep manoeuvring the drone (bMSC *Flying*). When a drone arrives at its destination, the pilot sends a landing command to the drone (bMSC *Landing*), which is acknowledged when the drone lands on the ground, and after it shuts down (bMSC *Shut Down*). If the battery is below the expected threshold, the drone performs a safe landing (bMSC *Safe Landing*) in accordance to its specification, and shuts down. The above scenario corresponds to the original behaviour of a drone, and represents its local requirements.

In order to *model exceptional scenarios*, the MSC specifications are amended with *interception points*. These interception points represent points in which conditions are analysed in

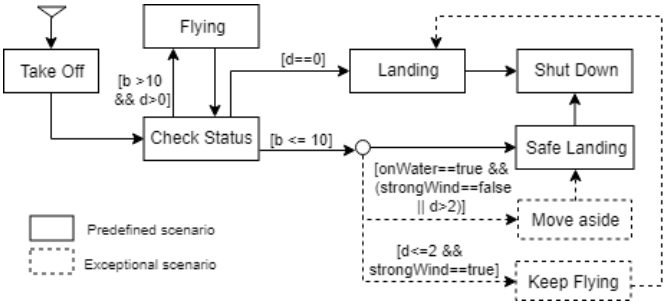

Fig. 2. hMSC for the drone payload delivery example

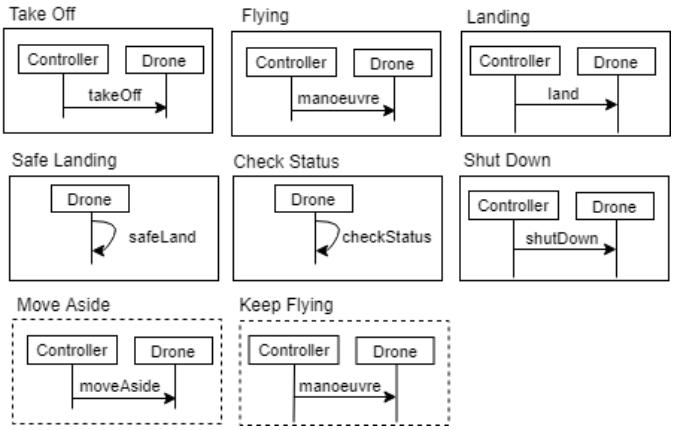

Fig. 3. bMSCs for the drone payload delivery example

order to decide whether the original expected scenario is executed or an introduced exceptional scenario has to take place. Several exceptional situations can be intercepted in the same point, indicating other possible behaviours according to alternative contexts. If there is no exceptional scenario in an interception point, the expected default scenario is performed. It may also be necessary to identify new contextual variables and conditions to be used in the transitions.

Exceptional scenarios are represented as dashed bMSCs in Figure 2. In our example, the exceptional scenarios added on the interception point state that if the distance to the expected destination is less than 2km and the wind is strong (condition *strongWind==true*), then the pilot can keep the drone flying (bMSC *Keep Flying*), even in a low-level battery situation, and land afterwards. In this case, the drone is able to complete its overall goal of delivering the organ payload successfully (global requirements GR1, above). In the case when the battery level is low, the drone is flying over the river (condition *onWater==true*), but it is more than 2km away or the wind is not strong, then the action is to move the drone aside (bMSC *Move Aside*) in order to land it safely on the ground and, therefore, satisfy the global requirement GR2.

After the normal and exceptional scenarios are modelled, the next step consists of automatically transforming the MSC specifications into a *parameterised Labelled Transition System* (pLTS). This is to allow for correctness checks of the model before and after adaptation, and to support the lack of knowledge of the ranges of values of contextual variables

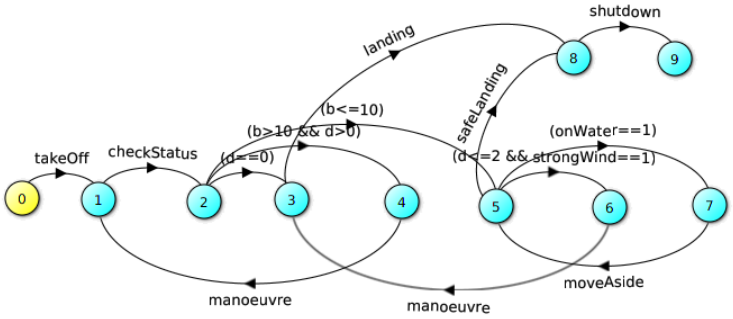

Fig. 4. Parameterised LTS illustrates the behaviours when the contextual variables are parameters bound to user specified constants.

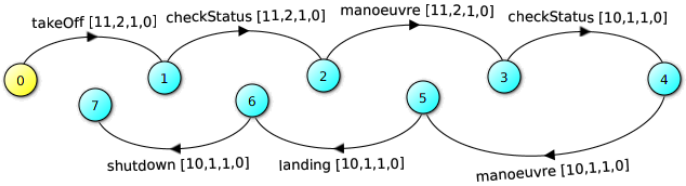

Fig. 5. LTS of Landing: the partial behaviour of landing bMSC, where the unnamed transitions are the transitions between different bMSCs; Through an ϵ-reduction, these transitions will be removed.

at this stage. The conditions on the transitions in the MSC specification are represented as guards on the state transitions. The approach uses an adaptation of the algorithm to generate LTS from MSC specifications proposed in [18], with the addition of the scenario transition guards in the resulting LTS. Figure 4 depicts the pLTS for the example of Figure 2.

After the parameterized LTS is generated, the values of the contextual variables can be instantiated by the developer, and a *context-specific LTS* representing the system behavior for those values is generated. Figure 5 illustrates the specific LTS generated when the developer sets the variables *b == 12%*, *d==2 km*, *strongWind==true*, and *onWater==true*. The variable values, in that order, are shown as parameters in the transitions in Figure 5. We assume that, after each drone movement, both battery level and distance are decreased by one unit. Steps one to four above were implemented as extensions of Lotus [7].

The context-specific LTS can be exported to LTSA [22] in order to check both local and global requirements specified in terms of properties. A *model checker* is applied to verify whether the participating components can satisfy global requirements of the system-of-systems, for which it was not initially designed. In the case when the component satisfies the requirements, it can be used as-is in the system-of-systems. However, when the component cannot satisfy global requirements (i.e., the case of a defiant component), a wrapper is specified in order to weave exceptional situations into the component specification in LTS, and it is verified by the LTSA model checker against the properties. The adaptation is considered *cautious* only when **all** properties (local and global) are satisfied by the component within the system-of-systems context. If the verification is considered successful through

model checking, the defiant behaviours need to be addressed further by design-time simulation.

B. Wrapper Design and Implementation

The wrapper design and implementation phase is composed of four steps, as shown in Figure 1. The first step consists of *identifying monitorable contextual variables* for analysing global requirements. In the payload organ delivery example, the monitorable contextual variables are: battery level, distance to destination, position of drones in relation to water, and wind condition. The contextual variables can be monitored by using either sensors on the environment of the system and on its participating components or external services such as a weather information web services.

In the next two steps the approach maps exceptional situations to corresponding concepts in the aspect-oriented paradigm (AOP) [13]. The main goal of the second step is to *analyse exceptional situations* and identify when they are triggered in the MSC specification in order to associate them with joinpoints.

The interception points in the MSC specifications represent the point in which exceptional situations should be analysed and they are mapped to joinpoints in AOP. For instance, considering our drone example, the only interception point is the one between the bMSCs *Check Status* and *Safe Landing*, when it is checked whether the battery level decreased to 10%. Subsequently, we have to identify joinpoints and define the respective pointcuts in which the code will be intercepted. For our example, we defined that method *safeLanding()* needs to be intercepted when it is called, in order to analyse exceptional situations.

In the third step, exceptions are addressed by *planning adaptation actions as advices* in the components. Using existing functionalities (e.g., maneuvering), while disabling certain functionalities (e.g., safe landing), it is possible to switch to a different solution through adaptation. These adaptions actions are implemented as *advices* in aspect-oriented programming language, with a type (before, after or around) dependant on the control flow of the exceptional situations.

The following code shows an excerpt of the *DroneAspect* implemented to wrap one defiant behaviour of the drone. We defined pointcut *checkExceptionalConditions()* that intercepts the call of the drone's original *safeLanding()* method (when the battery level reaches 10%).

```
1  public aspect DroneAspect {
2
3      pointcut checkExceptionalConditions():
4          call (void safeLanding());
5
6      void before(): checkExceptionalConditions() {
7          if (isOverWater()
8           && (getDistanceTargetHospital() >=2
9            || !isStrongWind()))
10             getDrone().moveAside();
11     }
12
13     void around(): checkExceptionalConditions() {
14         if (isOverWater()
15          && getDistanceTargetHospital() <=2
16             && isStrongWind())
17             getDrone().manoeuvre();
18     }
19 }
```

The above code shows two kinds of syntactic advices: *before* and *around*. The *before* clause (Line 6) is executed when the drone is over the water, and either the distance to the target hospital is no less than 2km or the wind is not strong (Lines 7-9). In this case, the drone executes a *modeAside()* method (Line 10), which moves the drone to fly over the ground and to execute the original *safeLanding()* method.

On the other hand, the *around* clause (Line 13) is executed when the drone is no more than 2km away from the destination, is flying over the water, and the wind is strong (Lines 14-16). Unlike the previous *before()* clause, new behaviour *manoeuvre()* (Line 17) replaces the *safeLanding()* method, which is no longer executed. When the drone reached the destination it performs the original *Landing()* method.

The implemented *wrappers* are deployed in the system so they can be used at runtime to adapt the behaviour of defiant components. This is done by weaving the wrapper aspects with the bytecode of the simulated system using the aspectJ compiler. If the defiant component is implemented in other programming languages, the wrapper aspects will need to be written in corresponding AOP languages [13].

C. Runtime Cautious Adaptation

The last phase encompasses the execution of the recently weaved system which, due to our wrapper solution, becomes a self-adaptive system. The steps of this phase are derived from the activities of the MAPE-K [20] control loop.

During its execution, the system keeps *monitoring contextual variables* by either checking internal resource levels or receiving environmental data provided by sensors. It may be necessary to execute some calculations in order to obtain the real value of the monitorable contextual variables. For instance, the value of variable distance from the target (d) is calculated by the difference between the GPS positions of the target hospital and current drone location.

After reading context information, the wrapper *analyses the exception situations* by intercepting the system execution in the defined pointcuts. In the example, this happens after the interception of *checkExceptionalConditions()* pointcut.

The last step corresponds to both Planning and Execution activities of the MAPE-K loop. When an exceptional condition is satisfied, the wrapper *applies the behaviour adaptation* by executing the exceptional scenario according to the rules implemented in the advices. The process continues to be executed until the system finishes its execution.

IV. Implementation and Evaluation

In order to evaluate our approach, we developed a prototype tool to simulate an extension of the payload organ delivery drone application described in Section I. We evaluated our approach for both single and multiple heterogeneous defiant components. In this section we present the evaluation setup

with its research questions and the simulator, and discuss the results of the evaluation and some threats to validity.

A. Evaluation setup

We evaluated the approach with respect to the correctness of the implementation of wrappers for identified defiant components to support satisfaction of global and local requirements. This is necessary since, although the properties of the wrappers have been model checked, it is still possible to introduce errors during the implementation of the scenario models. We also evaluated the approach with respect to the use of different wrappers against sources of exceptions generated by complex and uncertain environment. This evaluation includes variations in the multiplicity and precedence order, openness, and heterogeneity of defiant components in a system. For these evaluations we considered two research questions, namely:

- RQ1. Can the approach be used to achieve global requirements of a system-of-systems?
- RQ2: How does the approach behave with the increase of complexity in terms of multiplicity and precedence order, openness, and heterogeneity of defiant components?

We used the organ payload delivery application described in Section I and with the global requirements GR1 and GR2 described in Section III. For RQ1, we used the scenario shown in Figure 2. For RQ2, we used an extension of this scenario described in Figure 6. As shown in Figure 6, the extended scenario has three new functionalities, namely *Return to Home*, *Glide*, and *Call Rescue Boat*.

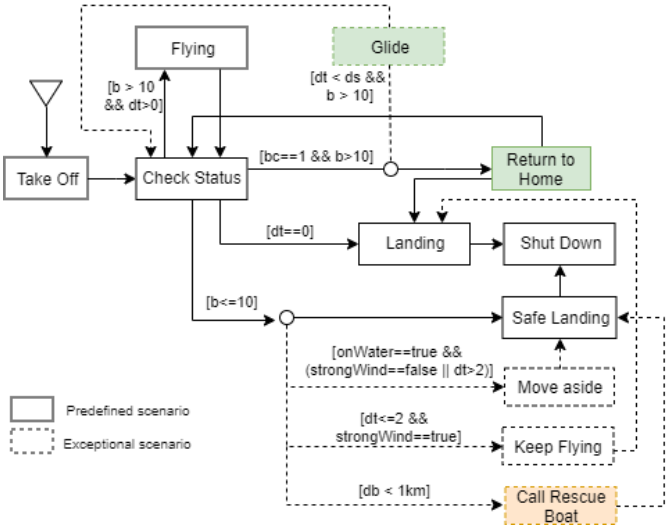

Fig. 6. Extended scenario (hMSCs)

The *Return to Home* functionality in a drone is executed when a drone bypasses a bad connection area and loses connection with the pilot. This is a common safety procedure present in the majority of existing drones. In order to simulate this situation, we introduced communication antennas in the scenario that emit periodical noise signals causing bad connection in congested areas. The exceptional scenario for this case consists of allowing the drone to *Glide* while waiting for the connection to return, maximising the chances of delivery. As shown in the figure, bMSC *Return to Home* happens when the contextual variable *bc==1* (bad connection) is true and the battery level is above 10% (*b>10*). When the drone arrives at the depart point (contextual variable *ds == 0*), it lands as usual. The exceptional scenario is represented by the bMSC *Glide*, which is performed when a bad connection is detected, the distance from the target (contextual variable *dt*) is less than the distance from the source hospital (contextual variable *ds*), and the battery level is greater than 10% ($b \geq 10$). In this case the drone *Glides* while waiting for a connection to be reestablished instead of automatically returning to home. The *Call Rescue Boat* functionality is due to an expansion of the application that supports the use of rescue boats to avoid drones landing on water. The *Call Rescue Boat* situation is triggered when the distance of the nearest boat to the drone (variable *db*) is less or equal to 1km. This case is described in Subsection IV-C for the *Openness* complexity.

For each experiment we used 100 executions with an original drone and a wrapped drone departing from the same place and with the same level of battery. The reason to choose 100 executions is to reach convergence and to get stable results in spite of the randomness of the executions. In fact, we observed that after 70 executions the results were already very stable, with less than $\pm 1\%$ perturbations. We also considered the same environmental conditions when performing the 100 executions.

In order to identify the level of battery for the drones to be evaluated, we considered nine intervals of batteries namely: 100%-91%, 90%-81%,...,20%-11% (a drone cannot take off with 10% or less battery, otherwise it will do a safe landing) and, for each interval, we run 100 drones with batteries in these intervals. We observed that drones with battery levels ranging from 61% to 80% were best suited to be used in the evaluation since most of the drones with battery level with more than 81% reached the destination, while most of the drones with battery level with less than 61% landed on water.

B. The simulator

We developed an open source drone simulator called Dragonfly [23] to implement both normal and exceptional situations of drones. Through its graphical user interface, the simulator allows the user to configure one or several drones, determine the source and target hospital locations; specify a river flowing between the hospitals; and set contextual variables such as initial battery level, battery consumption rate, wind strength, and drone above water. The simulator also offers an API to create programs to configure the environment, and plan and execute the journey of one or a fleet of drones. A snapshot of the simulator is depicted in Figure 7.

In the simulator, the user can manoeuvre the drones either manually or using an automatic pilot control, in which the drones try to reach the destination by executing a minimal distance routing algorithm. Moreover, the simulator supports the modelling of physical environment entities such as wind and rain, and uses heuristics of drone movements by taking

into account battery consumption, sensors, and the current status of the environment (e.g., the level of wind against a drone direction which may cause the drone to move slowly).

With the simulator, it is possible to track each drone by checking a panel that shows output traces of the drones in terms of performed scenarios, in accordance with the ones modelled in the hMSC. The panel also shows updates of the battery levels of the drones. It is also possible to change the wrapper implementation by a customised one and to reuse the same environment.

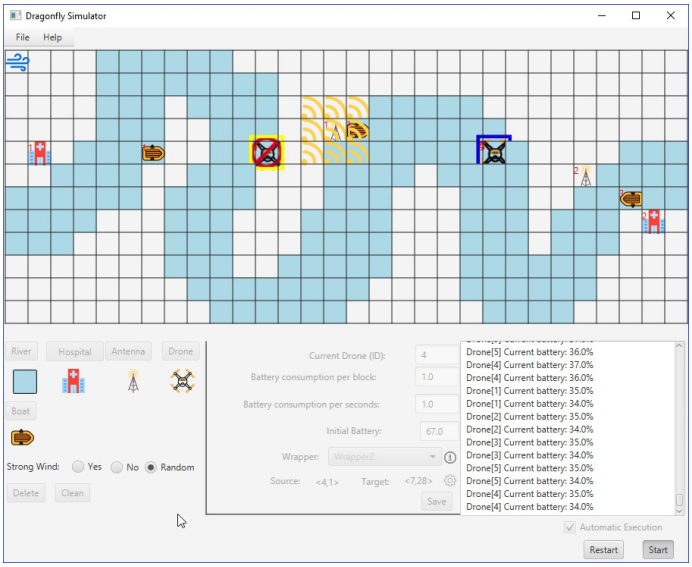

Fig. 7. A screenshot of the Dragonfly simulator

C. Experiments and results

In the following we present the experimental results to evaluate the two research questions.

RQ1. Can the approach be used to achieve global requirements of system-of-systems?

For this experiment we used the hMSC in Figure 2. We considered drones with only one defiant component concerning *Safe Landing* functionality when battery level is less or equal to 10%. We implemented a wrapper W1 representing the exceptional scenarios *KeepFlying* and *MoveAside*.

We executed three experiments for different wind strengths. We varied the frequency of the environmental variable *strongWind*, with respect to the chance of a strong wind happening. More specifically, we considered 10% (experiment E1), 50% (experiment E2), and 100% (experiment E3), respectively, for each second of time. In experiment E1, drones may benefit from strong winds to perform exceptional scenario *KeepFlying* only 10% of the time, while in E3 the wind is strong constantly. We modelled the scenario such that when a strong wind gust happens, it lasts for 2 seconds. Our objective with those experiments is to check if, as the strong wind turns more frequent, our approach generates better results for GR1.

Figure 8 shows the results for experiments E1, E2 and E3. For drones without wrappers (original drones) in which wind conditions and the distance to the hospital are not relevant, the percentage of drones that arrived at the destination (GR1) was 60% for E1 and E2, and 52% for E3. For global requirement (GR2), only 2%, 9%, and 5% of the original drones performed a safe landing on the ground (did not land on water) in experiments E1, E2 and E3, respectively. A high rate of drones landed on the water in the three experiments, ranging from 31% to 43% of drones. In these experiments the factors influencing the success of GR1 and GR2 are concerned with the amount of battery left in a drone to reach the destination and the place where the drones land in the case of safe landing, respectively.

The results in Figure 8 show that the cautious adaptation approach (the use of wrapper W1) improved the success rate, as the chance of strong winds increases. In the case of E1, for which the chance of a wind gust is low (only 10%), there was no change in the results for GR1: drones that reached the battery threshold less than 2km away of the target hospital performed a safe landing when there was no strong wind. On the other hand, our approach allowed 38 drones to land on the ground by performing the exceptional scenario *MoveAside*. This caused the results of succeeding with GR2 to increase from 2% to 40%. No drones landed on the water.

In experiment E2, seven drones were able to perform the exceptional scenario *KeepFlying*, which contributed to the rise of the success rate of GR1 from 60% to 67%, compared to E1. These numbers are even better for experiment E3, allowing 23 droned to perform *KeepFlying* and increasing the GR1 rate from 52% to 75%. Because of this, we note that less drones needed to perform *MoveAside* scenario when comparing the results for E1, E2 and E3. This explains the decrease of the success of GR2 rate from 40% (E1) to 33% (E2) and 35% (E3). Similarly to E1, no drone landed on the water in experiments E2 and E3. Our approach definitely supports the achievement of global requirements GR1 and GR2.

RQ2. How does the approach behave with the increase of complexity in terms of multiplicity and precedence order, openness, and heterogeneity of defiant components?

For this experiment we used the scenario in Figure 6. In order to evaluate RQ2, we considered three situations related to different complexity in the scenarios. We considered scenarios in which we varied the multiplicity and precedence order, openness, and heterogeneity of the defiant components in the drones, as discussed below.

Multiplicity and Precedence Order. In this case, we considered drones with two defiant functionalities, namely *Safe Landing* and *Return to Home*. We implemented a new wrapper W2 that extends wrapper W1 to deal with the new situation of *Return to Home*. For this case, the monitored environment variables are *bc* (bad connection), *dt* (distance from the target) and *ds* (distance from the source), and the *Glide* functionality is implemented as an *around* advice of the *Return to Home* functionality.

We executed experiment E4 with the same environment used in experiments E1 to E3, but with the addition of two

RESULTS OF SIMULATION FOR ORIGINAL DRONES										
Experiment	Drone	% Chance of Strong Wind	%Landed at Destination Normally	%Landed at Destination by Keep Flying	%Landed on the ground	%Landed on the ground after moving aside	%Landed on the Water	% GR1	% GR2	% LW
E1	Original	10	60	-	2	-	38	60	2	38
E2	Original	50	60	-	9	-	31	60	9	31
E3	Original	100	52	-	5	-	43	52	5	43
RESULTS OF SIMULATION FOR DRONES WITH WRAPPERS										
E1	W1	10	60	0	2	38	0	60	40	0
E2	W1	50	60	7	9	24	0	67	33	0
E3	W1	100	52	23	5	20	0	75	25	0

Fig. 8. Results for the experiments for RQ1

communication antennas that give noise signals, causing bad connection to the drones. A new version of an original drone with the *Return to Home* functionality and a drone with wrapper W2 were used in E4. In E4, we considered strong wind condition with 50% of chance.

Figure 9 shows the results for this new experiment. As can be seen from the results, due to communication antennas that may prevent the drones to arrive at the target hospital, only 12% of the original drones (without wrapper W2) landed normally at the destination (achieving GR1) when compared to experiments E1-E3, and 48 drones returned to home. Those drones, together with ten other drones that landed on the ground, resulted in only 58% of drones that satisfied GR2. Moreover, 30% of the drones fell on the river.

For the case in which wrappers are used, our approach helped to increase GR1 from 12% to 41% by allowing 14 drones perform the scenario *KeepMoving* (eight of these drones executed *KeepMoving* after performing the exceptional scenario *Glide*), and 15 drones glided for a while and resumed their journey to the target hospital. With respect to achieving GR2, 16 drones performed the scenario *MoveAside*. This allowed GR2 to increase from 58% to 59%. In this scenario, no drone landed on water.

Experiment E4 has shown that it is possible to have conflicts among defiant components and the need to impose a precedence order. For instance, there were situations in which a drone, during the execution of the exceptional scenario *Glide* due to a bad connection, had its battery level decreased to 10%, which triggered scenario *SafeLanding*. We used a prioritization approach in which the safe landing functionality has higher precedence over the gliding functionality, since a low battery situation is more critical and, not dealing with it, can cause possible hazards to the drone and to the payload. In Figure 9, column "Gliding and Landed at Destination by Keep Flying" indicates the percentage of times that the above mentioned situation occurred. As can be seen, with this strategy, our approach was able to avoid losing from 30 to 43 drones in experiments E1-E4.

Openness. In this case, we wanted to evaluate how the approach supports the introduction of other participating components in the system-of-systems and continues to deal with defiant situations. We expanded the organ payload delivery application to support the participation of rescue boats, which allows drones to land on boats, instead of landing on water. In this case, we added a boat centre with three rescue boats to support the rescue of drones. In this scenario, during the *Safe Landing* procedure, if a drone does not satisfy environmental conditions to perform the exceptional scenario *KeepFlying*, then it sends an *S.O.S. message* to the boat centre, which knows where all rescue boats are along the river. The boat centre verifies which boat is closer to the drone to be rescued and, if it is within the distance threshold of 1 km [3], the boat center sends the boat to rescue the drone. After helping the drone, the rescue boat takes the drone with the payload to the destination hospital. In this way, one can increase the success rate of GR1. In the case in which there are no rescue boats available at an appropriate distance to the boat, the drone attempts to perform the *MoveAside* exceptional scenario.

We implemented a new wrapper W3 as an extension of wrapper W2 adding the handling of S.O.S. messages. The above situation was executed in experiment E5 and the results are shown in Figure 9. In this case, the achievement of GR1 increases to 57% as 16 drones were rescued by boats and delivered to the hospital. In the cases in which there was no boat near to those drones, then the drones executed scenario *MoveAside*. This explains why the result for GR2 decreased for this experiment, when compared to E4 and W2, from 59% to 43%. No drone was lost by landing on the water.

Heterogeniety. In this case, we wanted to evaluate the approach for heterogeneous defiant components. More specifically, we analysed the case in which defiant situations occur for different types of components in a system-of-systems. We expanded the boat rescue scenario described above to allow other types of boats such as cargo transportation boats that are in the river with a purpose different from that of the organ payload delivery application.

Consider the situation in which transportation boats can receive requests from the boat centre to rescue drones, but can only do this if they are free (i.e., a boat is not doing a transportation service). Assume that the boat controller's code is not accessible. When a boat starts a transportation service, it becomes unavailable for rescuing drones, refusing to respond

[3]We used a distance threshold of 1 km to give enough time for the boat to arrive to rescue a drone before the drone lands on water.

RESULTS OF SIMULATION FOR EXPERIMENTS E4, E5 AND E6															
Experiment	Drone or Boat	%Landed at Destination Normally	% Landed at Destination by Keep Flying	%Landed on the ground	%Landed on the ground after moving aside	%Landed on the Water	%Returned to Home	%Glided and Landed at Destination Normally	%Glided and Landed at Destination by Keep Flying	%Recovered and Delivered by a Boat	%Recovered and Delivered by a Defiant Boat	%GR1	%GR2	%LW
E4	Original	12	-	10	-	30	48	-	-	-	-	**12**	**58**	**30**
E4	W2	12	6	8	16	0	35	15	8	-	-	**41**	**59**	**0**
E5	W3	12	6	8	0	0	35	15	8	16	-	**57**	**43**	**0**
E6	W4	16	5	15	0	0	26	14	4	2	18	**59**	**41**	**0**

Fig. 9. Results for the experiment for RQ2

to new requests from the boat centre until its transportation task is completed. A conflict exists between the boat's local requirement of being unavailable due to a transportation service and the system-of-systems global requirement of a drone having to arrive at a hospital. This conflict occurs only in the exceptional scenario where the drone needs to make an emergency landing while flying over the river, and there is a boat close enough to make the rescue. However, the boat is already engaged in a service. Therefore, both the boat and the drone are considered defiant components.

In this case, it is necessary to implement two wrappers: one for the drone and one for the boat. We considered wrapper W3 for the drone and implemented a new wrapper W4 for the boat. Wrapper W4 changes the boat's destination to the location of a drone that needs to be rescued. After rescuing the drone, the boat will restore to its normal service.

Figure 9 shows the results for experiment E6 in which we considered drones with wrapper W3 and boats with wrapper W4. The results show that five drones arrived at the destination by performing *KeepFlying* functionality, 18 drones glided and reach the target hospital, and two drones are recovered by an available boat. In this new situation, 18 drones were rescued by a defiant boat, which means that if those boats did not implement the wrapper, these drones would have performed a *MoveAside* functionality (as specified in W3). The approach avoided drones to land on water and helped deliver the payloads to their destinations. It contributed to the rise of satisfying GR1 to 59%. In fact, when comparing the results for experiments E4, E5 and E6, the latter is the one with a higher achievement for GR1. The results also show that 41% of the drones satisfied GR2, and no drone landed on water.

D. Discussions

The experiments showed that our cautious adaptation approach helps improve the achievement of global requirements in a system-of-systems. We are aware of threats to the validity of our evaluation and limitations of the proposed approach. We discuss them below.

Threats to validity. There are three types of threats to validity in our evaluation.

Construct validity. We simulated up to 100 executions in different scenarios and considered both homogeneous and heterogeneous defiant components. However, in real-life cases, the contextual variables and their combinations can be much more complex. Moreover, we only simulated the applications in the context of system-of-systems, but defiant components can be found in other domains such as microservice-based distributed systems and Internet of Things. In our future work we plan to test the cautious adaptation approach in larger case studies involving a higher number of defiant components and considering other application domains.

Internal validity. The choices made during the simulation such as the environment design, distance between the hospitals, drone speed, and location of antennas may affect the results. Considering all possible choices at once is hard since the combination of variables may lead to an explosion of possibilities. However, in the evaluation, we tried to make reasonable realistic choices for the simulations, considering plausible values for the environmental variables and internal drone components such as battery levels.

External validity. Defiant components by our definition are a fairly generalisable concept which may manifest as physical hardware components (like the drones in this paper), as well as cyber-physical components including human in the loop. In the future we plan to evaluate the approach including scenarios with defiant cyber-physical components.

Limitations. Our approach extends traditional adaptation approaches by changing parameters or calling functions of components that do not offer such possibilities when the source code of the components are not available. In a way, the assumption of adaptability is exchanged with a weaker assumption – that it is possible to define *joinpoints* at which to intervene. While this works well in highly modularised systems with clear separation of concerns (as in the examples shown), this may pose further challenges to complex legacy software, which is left for future investigations.

Another limitation is with regards to the choice of techniques to implement wrappers. Although AOP is shown directly applicable to defiant components in this work, we are aware of other wrapper techniques which might be more suitable for other types of defiant components in domains such as self-driving cars and driverless trains. We plan to investigate this in future work. The approaches works without access to the source code of defiant components. However, it is still necessary to know the API signatures of internal methods and to have access to the executable or the byte code part of the implementation. For instance, considering our example, it is necessary to know *manoeuvre()* and *checkStatus()*. In the experiments we only used aspect-oriented programming for Java programs through AspectJ. The application of AOP paradigm to other programming languages will be investigated

in the future.

Although we have implemented the entire process, the wrappers have not been deployed to drones in the real world, which would require additional approvals by legal and ethics regulatory bodies. Moreover, the use of a simulator abstracted away the time necessary to perform adaptations, which may be in fact critical in the context of fast-flying drones. To mitigate this threat, we are currently liaising with industrial organisations (e.g., local transport company and real hospitals) to deploy the solution into their system-of-systems. In the future, we would also like to analyse the delays that may be caused by computation and communication activities.

V. Related work

Our cautious adaptation approach is related to works in the following areas: (i) identification of normal and exceptional scenarios, (ii) self-adaptive systems and runtime obstacles, (iii) system-of-systems and COTS adaptation, (iv) aspect-oriented programming and dependency injection, and (v) safety assurance of drones and simulation. We discuss below existing works in each of these areas.

Identification of normal and exceptional scenarios. The first steps of our approach consist of modelling both normal and exceptional situations of a system-of-systems as scenarios. Although the choice is left to the adopters of our approach on how scenarios will be identified, we envisage that some techniques can aid software engineers in this task such as self-adaptive requirements modelling [24], specification mining [25], and reverse engineering [26] [27] [28] techniques.

The work in [24] uses an adaptation-oriented requirement modelling approach for system-of-systems based on Goal-Oriented Requirements Engineering (GORE) concepts [29]. The approach allows to specify scenarios for adaptation strategies. This approach could be helpful to model normal and exceptional MSCs used in our approach.

A specification miner takes a program presented as a set of static or dynamic traces as its input, and produces one or more candidate specifications with respect to a set of interesting program events [30]. In this direction, one could propose a specification miner that receives dynamic traces from the system-of-systems execution and generates a set of initial scenarios, similarly to the approach proposed in [25]. Other scenarios could also be discovered using reverse engineering techniques that process execution traces and produce candidate MSCs [26]. The work in [27] proposes to reverse engineer goals directly from legacy source code through refactoring-based approaches. The work in [28] uses mined message sequence charts from the partial ordering of execution traces of concurrent programs. This technique could be used to support the situation in which MSCs of various components need to be composed concurrently.

Self-Adaptive Systems and Runtime Obstacles. In order to provide high-assurance and management of uncertainty at runtime [31], [32], self-adaptive systems typically use MAPE-K control loops [20] based on previously known requirements [33] and predefined high-variability alternatives [34], [35]. To manage inconsistencies between global requirements and defiant behaviours of local components, we propose a cautious adaptation approach where scenarios are used to identify contextual variables to monitor and points of interception. In this way, the need to prepare high-variability alternatives is replaced by creating wrappers that handle exceptional scenarios and contextual conditions.

Traditionally safety requirements [36] can be analysed to identify contingency requirements using obstacle analysis [37], where contingency requirements can be viewed as exceptional conditions, but they are exposed in requirement goals, rather than in concrete scenarios at runtime. With a similar motivation to handle exceptions at runtime, Cailliau et al. [38] proposed probabilistic obstacles that can be addressed through runtime monitoring and verification. Although probabilistic obstacles are conceptually similar to defiant situations, our approach directly modifies the behaviour of defiant components-off-the-shelf (COTS), without resorting to probabilistic evidence collection and reasoning on probabilistic LTL.

System-of-Systems (SoSs) Inconsistency and COTS Adaptation. One of the problems to manage inconsistency of SoSs is to identify conflicts among different components [39] and resolve conflicts through the use of a utility function by the MAPE-K self-adaptive feedback loops [40]. Defiant components are a source of conflicts and our cautious adaptation approach can be seen as a means to resolve conflicts by design.

When global requirements are not achievable by the components off-the-shelf (COTS), adaptation would be required [41]. Although COTS adaptation has similar motivations to our cautious adaptation approach, the main difference lies in the support of multiple defiant components within COTS. The proposed wrappers aim to partition defiant components by the internal behaviours rather than traditional adaptation at component interfaces.

AOP and DI. Modification of legacy systems can be supported by the use of aspect-oriented programming (AOP) [42] and dependency injection (DI) [16], in particular when it is not allowed to change the design of original system intrusively. Both AOP and DI could modify the behaviour of original programs. AOP can be applied to both source and binary code, while DI is applicable typically when the programming languages support reflection [43]. In our work, AOP was used for implementing the adaptation concept. Compared to a simple application of AOP, our approach introduces additional "caution" by checking, proactively, that both local and global requirements of the system are satisfied for normal and exceptional conditions of the environment. When the components are high-level (rather than simulated programmatically in the running example), requirement-level aspect-oriented approach may also be considered [44], so that the advices can be implemented using traditional functions.

Safety Assurance of Drones and Simulation. Formal methods have been applied to provide assurance for safety requirements satisfaction for UAVs [45]. However, runtime assur-

ance of required properties can be weakened by exceptional conditions. Scenarios have been applied to address these exceptional conditions formally. However, such application of scenarios was not enforced by runtime systems due to the black box nature of legacy systems. Our approach makes the early connection between scenario-based assurance of formal properties and their practical enforcement through wrappers.

Safety requirements, among many functional and non-functional requirements, are critical for transportation systems such as drones [46]. Although several attempts exist for safety assurance, incidents of drones still happen, often leading to interference in other transport systems like passenger aircraft and airports. In such cases, the protection of drones can be seen as a rather narrowed view of safety with respect to a broader, and much more challenging, safety and other critical requirements for the system-of-systems. The running example of payload organ delivery illustrates that it is not sufficient to consider just the safety cases of the drones. Self-adaptive system-of-systems have to address global requirements.

While a swarm of drones are difficult to be evaluated, simulation is a recognized way to assure functionalities before physical tests in the sky. Drone simulations such as Dronology [47] are an established way to simulate drones for critical safety properties such as separation of distances. Our simulator goes beyond safety-critical properties and consider more general requirements of system-of-systems.

VI. Conclusion and Future Work

In this paper we proposed a novel approach to support changes in participating components in system-of-systems that have been designed to satisfy predefined requirements, and not necessarily intended to change their behaviour in order to support global requirements of system-of-systems. We call these *defiant components*. We presented a *cautious adaptation* approach that supports changes in the behaviour of defiant components in order to satisfy global requirements in exceptional situations.

The cautious adaptation approach guarantees that changes will not interfere with the original functionalities of the participating components under normal situations. The approach uses scenarios to represent normal and exceptional situations, uses wrappers implemented in aspect-oriented techniques to represent behaviour of exceptional situations, and executes runtime adaptation. We use an example of a payload organ delivery drone application to illustrate and evaluate the work.

Currently, we are extending the approach to support on-the-fly identification of new exceptional situations due to emergent behaviours, and the creation of their respective wrappers. We are also evaluating the scalability of the work with respect to large-scale scenarios in different domains, and considering components with complex requirements, functionalities, and prioritisation decisions.

We plan to define a reference architecture for runtime adaptation of defiant components, which can be applied to multiple application domains. We envisage extending the identification of exceptional scenarios to deal with emergent scenarios at runtime. Another area for future work is concerned with the extension of the approach to support analysis of the impact of adaptation during systems execution with respect to multiple crosscutting aspects (e.g., performance), and in terms of its side-effect on other components participating in a system-of-systems.

VII. Acknowledgments

This work is partially supported by CNPq/Brazil under grant Universal 438783/2018-2 and Funcap/Brazil under grant UKA-0160-00005.01.00/19; EPSRC Platform Grant on Secure Adaptable Usable Software Engineering (EP/R013144/1); EU H2020 SESAR EngageKTN on DroneIdentity (No. 783287); ERC Advanced Grant on Adaptive Security and Privacy (No. 291652), and Science Foundation Ireland (SFI) grant 13/RC/2094.

References

[1] M. W. Maier, "Architecting principles for system of systems," vol. 1, 01 1998.

[2] V. E. Silva Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos, "Awareness requirements for adaptive systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '11. New York, NY, USA: ACM, 2011, pp. 60–69. [Online]. Available: http://doi.acm.org/10.1145/1988008.1988018

[3] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 341–350. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985840

[4] D. Weyns and J. Andersson, "On the challenges of self-adaptation in systems of systems," in *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*, ser. SESoS '13. New York, NY, USA: ACM, 2013, pp. 47–51.

[5] R. de Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. R. Schmerl, D. Weyns, L. Baresi, N. Bencomo, Y. Brun, J. Cámara, R. Calinescu, M. B. Cohen, A. Gorla, V. Grassi, L. Grunske, P. Inverardi, J. Jézéquel, S. Malek, R. Mirandola, M. Mori, H. A. Müller, R. Rouvoy, C. M. F. Rubira, É. Rutten, M. Shaw, G. Tamburrelli, G. Tamura, N. M. Villegas, T. Vogel, and F. Zambonelli, "Software engineering for self-adaptive systems: Research challenges in the provision of assurances," in *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*, 2013, pp. 3–30.

[6] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Softw. Syst. Model.*, vol. 15, no. 1, pp. 31–69, Feb. 2016.

[7] D. M. Barbosa, R. G. de Moura Lima, P. H. M. Maia, and E. C. Junior, "Lotus@Runtime: A tool for runtime monitoring and verification of self-adaptive systems," in *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 24–30.

[8] P. Arcaini, E. Riccobene, and P. Scandurra, "Formal design and verification of self-adaptive systems with decentralized control," *ACM Trans. Auton. Adapt. Syst.*, vol. 11, no. 4, pp. 25:1–25:35, Jan. 2017.

[9] T. Viana, A. Zisman, and A. K. Bandara, "Identifying conflicting requirements in systems of systems," in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, Sep. 2017, pp. 436–441.

[10] K. YoungGab and C. Sungdeok, "Threat scenario based security risk analysis using use case modeling in information systems," *Security and Communication Networks*, vol. 5, no. 3, pp. 293–300, 2011.

[11] S. Uchitel, D. Alrajeh, S. Ben-David, V. Braberman, M. Chechik, G. De Caso, N. D'Ippolito, D. Fischbein, D. Garbervetsky, J. Kramer, A. Russo, and G. Sibay, "Supporting incremental behaviour model elaboration," *Computer Science - Research and Development*, vol. 28, no. 4, pp. 279–293, Nov 2013. [Online]. Available: https://doi.org/10.1007/s00450-012-0233-1

[12] X. Ban and X. Tong, "A scenario-based information security risk evaluation method," *International Journal of Security and Its Applications*, vol. 8, no. 5, pp. 21–30, 2014.

[13] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 313–, Sep. 2001. [Online]. Available: http://doi.acm.org/10.1145/503271.503260

[14] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd ed. Wiley Publishing, 2006.

[15] D. Xu, I. Alsmadi, and W. Xu, "Model checking aspect-oriented design specification," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1, July 2007, pp. 491–500.

[16] M. Fowler, "Inversion of control containers and the dependency injection pattern," http://www.martinfowler.com/articles/injection.html, 2004, accessed: 2015-07-23.

[17] M. Wermelinger and Y. Yu, "Analyzing the evolution of eclipse plugins," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 133–136. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370783

[18] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Trans. Softw. Eng.*, vol. 29, no. 2, pp. 99–115, Feb. 2003. [Online]. Available: https://doi.org/10.1109/TSE.2003.1178048

[19] D. Harel and P. S. Thiagarajan, "Message sequence charts," in *UML for Real: Design of Embedded Real-Time Systems*, L. Lavagno, G. Martin, and B. Selic, Eds. Boston, MA: Springer US, 2003, pp. 77–105.

[20] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., Jun. 2005.

[21] M. A. Jackson, *Problem Frames - Analysing and Structuring Software Development Problems*. Pearson Education, 2000.

[22] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd ed. Wiley Publishing, 2006.

[23] P. H. Maia, L. Vieira, M. Chagas, Y. Yu, A. Zisman, and B. Nuseibeh, "Dragonfly: a tool for simulating self-adaptive drone behaviours," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2019, pp. 107–113.

[24] M. Maciel, P. H. Maia, F. C. M. B. Oliveira, and F. Maciel, "Adore: An adaptation-oriented requirement modeling approach for systems of systems," in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM, September 2019.

[25] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," *SIGPLAN Not.*, vol. 37, no. 1, pp. 4–16, Jan. 2002. [Online]. Available: http://doi.acm.org/10.1145/565816.503275

[26] F. C. d. Sousa, N. C. Mendonca, S. Uchitel, and J. Kramer, "Detecting implied scenarios from execution traces," in *Proceedings of the 14th Working Conference on Reverse Engineering*, ser. WCRE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 50–59.

[27] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite, "Reverse engineering goal models from legacy code," in *13th IEEE International Conference on Requirements Engineering (RE 2005), 29 August - 2 September 2005, Paris, France*, 2005, pp. 363–372.

[28] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, "Mining message sequence graphs," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 91–100.

[29] A. Van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, ser. RE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 249–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882477.883624

[30] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 461–476.

[31] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for self-adaptive systems," B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26.

[32] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly, "Engineering trustworthy self-adaptive software with dynamic assurance cases," *IEEE Trans. Software Eng.*, vol. 44, no. 11, pp. 1039–1069, 2018.

[33] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy goals for requirements-driven adaptation," in *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference*, ser. RE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 125–134.

[34] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos, "Requirements-driven design of autonomic application software," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '16. Riverton, NJ, USA: IBM Corp., 2016, pp. 23–37.

[35] M. Salifu, Y. Yu, and B. Nuseibeh, "Specifying monitoring and switching problems in context," in *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India*, 2007, pp. 211–220.

[36] R. R. Lutz, "Software engineering for safety: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 213–226.

[37] R. R. Lutz, A. Patterson-Hine, S. Nelson, C. R. Frost, D. Tal, and R. Harris, "Using obstacle analysis to identify contingency requirements on an unpiloted aerial vehicle," *Requir. Eng.*, vol. 12, no. 1, pp. 41–54, 2007. [Online]. Available: https://doi.org/10.1007/s00766-006-0039-4

[38] A. Cailliau and A. van Lamsweerde, "Runtime monitoring and resolution of probabilistic obstacles to system goals," in *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, 2017, pp. 1–11.

[39] T. Viana, A. Zisman, and A. K. Bandara, "Identifying conflicting requirements in systems of systems," in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, Sep. 2017, pp. 436–441.

[40] T. Viana, A. Zisman, and A. K. Bandara, "Towards a framework for managing inconsistencies in systems of systems," in *Proceedings of the International Colloquium on Software-intensive Systems-of-Systems at 10th European Conference on Software Architecture*, ser. SiSoS@ECSA '16. New York, NY, USA: ACM, 2016, pp. 8:1–8:7.

[41] D. Wile, R. Balzer, N. Goldman, M. Tallis, A. Egyed, and T. Hollebeek, "Adapting cots products," 10 2010, pp. 1 – 9.

[42] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," in *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, 2001, p. 313.

[43] S. Chiba and R. Ishikawa, "Aspect-oriented programming beyond dependency injection," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 121–143.

[44] Y. Yu, J. C. S. do Prado Leite, and J. Mylopoulos, "From goals to aspects: Discovering aspects from requirements goal models," in *12th IEEE International Conference on Requirements Engineering (RE 2004), 6-10 September 2004, Kyoto, Japan*, 2004, pp. 38–47.

[45] M. Webster, M. Fisher, N. Cameron, and M. Jump, "Formal methods for the certification of autonomous unmanned aircraft systems," in *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security*, ser. SAFECOMP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 228–242. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041619.2041644

[46] C. Lin, D. He, N. Kumar, K. R. Choo, A. Vinel, and X. Huang, "Security and privacy for the internet of drones: Challenges and solutions," *IEEE Communications Magazine*, vol. 56, no. 1, pp. 64–69, Jan 2018.

[47] J. Cleland-Huang, M. Vierhauser, and S. Bayley, "Dronology: an incubator for cyber-physical systems research," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 109–112.